Parallel Computing - Final Report (WA1,WA2,WA3)

Optimising a molecular dynamics' simulation code

October 23, 2023

José Fonte
Universidade do Minho
a91775

Bernard Georges
Universidade do Minho
pg53698

Abstract

This assignment focus on three differnt stages of optimising a simple molecular dynamics' simulation applied to atoms of a given gas. The work group used several optimization techniques to decrease the inital running time from *250 secs* to a best of *2.35 secs*. The following report explains the steps and rationale of the group that led to 99.06% reduction over the original running time.

1 | Introduction

The following report describes the work developed by the team, during the semester, optimising an inneficient program provided by the Professors. The given code is a simple molecular dynamics' simulation applied to atoms of a given gas - it uses Newton's Law and Lennard Jones potential to describe the interactions among two particles (force/potential energy).

The report details the work developed by the group in three different phases of optimization:

- Phase 1 - Code Optimizations: where we profile, analyze, and optimize the codebase given, applying multiple code optimization techniques taught.
- Phase 2 - OpenMP : where we implement the OpenMP API to parallelize the code on the CPU.
- Phase 3 - CUDA : where we implement the CUDA API to parallelize the code on an accelerator (Nvidia GPU).

2 | Phase 1 - Code Optimizations

During this phase the main goal was to lower the running time with techniques taught in class. As a Benchmark, we have the CPU_time given by the formula,

$$\mathrm{CPU}_{\mathrm{time}} = \#\mathrm{Inst} * \mathrm{CPI} * \mathrm{Clk}_{\mathrm{cycles}}$$

where, to have a lower running time, the group can decrease the number of instructions and/or decrease the *CPI* because the Clock Cycles is imposed by the CPU.

2.1 | Optimization journal

The methodology followed by the group was of continuous analisys and optimization, the following chapters describe the process the group went through to achive the best running time of *5.35 secs*.

2.1.1 | Big O | Code Analysis

The Big O analysis enables us to identify the space and time complexity of a piece of code and do further optimizations if possible. Throughout our analysis, the group noticed that various loops complexity could be reduced using well-know Loop-Level Parallelism techniques such as Loop Unrolling and some flags mentioned in 2.4 | Flags Used.

Potential Improvements :

- *potential()* : $O(3n^2) \rightarrow O(n^2)$
- *computeAccelerations()* : $O(6n^2) \rightarrow O(n^2)$
- *velocityVerlet()* : $O(7N) \rightarrow O(3N)$
- *initializeVelocities()* $O(15N) \rightarrow O(3N)$

2.1.2 | Loop Unrolling

Loop unrolling executes multiple loop iterations in a single pass, leading to an increase in performance due to:

- reduced loop overhead - lowering conditionals jumps, control variables increment and mispredictions in jumps.
- increased LLP - allowing the processor to execute instructions concurrently every pass taking full advantage of multiple execution pipelines.

The implementation of this technique throughout the code can be seen in attachments : Listing 1 and Listing 2, which shows the code before and after loop unrolling in the *potential()* function.

Even though none of the improvements reduce the complexity in an order of magnitude (*E.g. :* $O(n^3) \rightarrow O(n^2)$), the way the code works makes it that a small improvement has a compounding effect, leading to major improvements, better described in "Results Comparison".

2.2 | Gprof Results | Profilling

The *gprof* tool provides a detailed breakdown of where the majority of CPU cycles are being consumed, allowing us to pinpoint were our atention to optimization needs to be.

As shown in Figure 11 our main focus at thatmoment had to be the *Potential()* function.

2.2.1 | Simplifying Math Functions

After analising the function code the group noticed that the math used could be simplified.

$$\begin{cases} \mathrm{rnorm} = \sqrt{\mathrm{r2}} \\ \mathrm{quot} = \frac{\mathrm{sigma}}{\mathrm{rnorm}} \end{cases} \Rightarrow \begin{cases} \mathrm{quot} = \frac{\mathrm{sigma}}{\sqrt{\mathrm{r2}}} \end{cases}$$

$$\begin{cases} \mathrm{term2} = \mathrm{quot}^6 \\ \mathrm{term1} = \mathrm{quot}^{12} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^6} \\ \mathrm{term1} = \frac{\mathrm{sigma}^{12}}{\sqrt{\mathrm{r2}}^{12}} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \frac{\mathrm{sigma}^{12}}{\mathrm{r2}^6} \end{cases}$$

$$\Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} * \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \mathrm{term2} * \mathrm{term2} \end{cases}$$

Using the equations presented we can simplify the code in Listing 3 to the one in Listing 4. This procedure helped reduce the amount of multiplications and removed the `sqrt()`and `pow()` functions which take a heavy toll on the ALU performance. As mentioned previously, because of the compounding effect of this optimization and all others, these small changes had a big consequence on the #Inst of our program. Besides that calling external libraries can cause various overhead because of data transfer serialization and the function call overhead.

- *transfer serialization*: when your program needs to pass large data structures or complex objects to external library functions, the serialization and deserialization processes can be computationally expensive.

With this changes inplace the group, once again, used the *gprof* tool that now indicated *computeAccelerations* as the function to tackle next.

2.2.2 | Matrixes Analysis | Code Analysis

After analysing the matrixes the group noticed that some functions run similar workloads, creating a large amount of repeated instructions over the program's execution time. Therefore the functions *computeAccelarations()* and *potential()* were joined into *joinedPotentialComputeAcc()* and the functions Kinetic() and *MeanSquaredVelocity()* were joined into *joinedKineticMSV()*.

2.2.3 | Functions Fusion

The first coupling *joinedPotentialComputeAcc()* was due to both acessing the matrix `r[i][j]`. Some small changes were made to the code to accomodate this fusion, the one that should be noticed is *Potential()* loop was changed to avoid doing double the calculations, something the group noticed on analysis - this means the nested for loop now goes from $i \rightarrow N$ and the potential doubles so it is $\text{PE} += 8 * \text{epsilon} * (\text{term1} - \text{term2})$.

The second pairing *joinedKineticMSV()* was due to both acessing the matrix `v[i][j]` and doing some calculations after. Another reason was because there was data independency which means their variables don't need the others calculations which enables parallel execution.

The couplings helped reduce the amount of times the code needed the matrixes and the amount of loop needed to do so by calculating shared values that in the end were used in equations to generate the goal. This helped in reduce the loop overhead, amount of memory access and computation of the equations by atleast N times for each fusion being the fusion of the *Potencial* and *computeAccelerations* the most noticable of all the changes.

2.2.4 | Flags Used

The last step of optimization was implementing a few flags so the compiler produces better machine code, each one of these plays a role in optimizing and enhancing the performance of code, making it more efficient for specific hardware.

- `-Wall` : used to enable a comprehensive set of compiler warning messages, such as uninitialized variables, unused variables, logical errors and other problems.
- `-pg` : required by *gprof* which is a profilling tool that allows you to gather performance data about your program.
- `-O3` : flag designed to maximize the execution speed of your code by applying aggressive optimization techniques, it checks all optimization flags available, including `-ftree-vectorize` that enables the compiler to vectorize loops and utilize SIMD instructions.
- `-msse4` : specifies that the compiler should generate code using SSE4 instructions which allows operations with 128bits vectors.
- `-mavx` : instructs the compiler to generate code that uses AVX instructions, which offer even more advanced SIMD capabilities.
- `-mfpmath=sse` : specifies that the compiler should use SSE math instructions for floating-point operations.
- `-march=x86-64` : sets the target architecture to x86-64 (64-bit x86 architecture), optimizing code generation for 64-bit systems.

2.2.5 | Common Compiler Optimizations

Some Common Compiler Optimizations like Loop Unrolling, Fusion & Reduction in Strength were already mention in there previous contexts but other optimizations were also implemented such as Dead Store Elimination & Constant folding.

2.2.5.1 | Dead Store Elimination

Due to the -Wall compiler flag and some code analysis, multiple variables were flaged as being unused, being some examples of the ones removed:

- `trash[10000]`
- `F[MAXPART][3]`
- `*infp`
- all the loose variables from loop unrolling
- reducing number of chars in `prefix[1000], tfn[1000], ofn[1000], afn[1000]` to 32 chars.

2.3 | Results Comparison

At this final stage the group decided on making some final observations about the improvents in each phase of development using the *srun* tool to show the time, instructions and clock cycles for every iteration.

2.3.1 | Loop Unrolling

```
1,174,118,004,671      instructions              #    1.52  insn per cycle
  774,689,395,085      cycles

    258.357289771 seconds time elapsed

    258.360590000 seconds user
      0.001000000 seconds sys
```

Figure 1: Result after loop unrolling

2.3.2 | Symplifying Math Functions

```
226,804,957,546      instructions              #    1.91  insn per cycle
119,034,908,953      cycles

    38.884537768 seconds time elapsed

    38.878885000 seconds user
     0.002000000 seconds sys
```

Figure 2: Result after symplifying math functions

2.3.3 | Functions Fusion

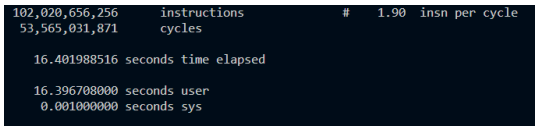

Figure 3: Result after functions fusion

2.3.4 | Final Result with Flags Used

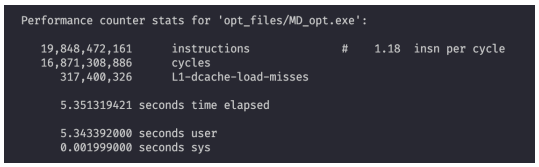

Figure 4: Final Result with all optimizations

2.4 | Conclusion Phase 1

On the first phase, we believe the results were good and we are very happy with the amount we learned and improved in this part of project. We particularly enjoyed learning about the compiler optimization techniques as well as avoiding data dependencies. Looking back, we believe that we could have explored more of code vectorization and further improvements through the analysis of the assembly code.

3 | Phase 2 - Implementing OpenMP

In the seconde phase of the project, the group focused on further optimization leveraging the power of OpenMP, a multiprocessing programming interface that facilitates the integration of threads into the code designed sequently in the previous stage.

As a baseline of performance we have the code developed in the previous phase of the project, which follows a sequencial execution. As seen in Figure 13, the execution time is around 70 seconds, therefore in order to improve it the group followed this workflow:

- identify the application hot-spots (code blocks with high computation time);
- analyse and present the alternatives to explore parallelism within the hot-spots identified;
- select an approach to explore parallelism, justified by a scalability analysis;
- implement and optimise the approach;
- measure and discuss the performance of the solution.

3.1 | Identifying Hot Spots

The initial phase involves pinpointing the critical areas of the code that necessitate improvement and in order to achive this the group employed the “perf” profiling tool. Following the execution of the sequential version of the code, the profilling results, shown in Figure 12, directed our attention to the crucial bottleneck — the *“joinedPotentialComputeAcc()”* function.

3.2 | Analising Parallelism Alternatives

As denoted in Listing 5, the sequential code within the hot-spot identified presentes three *for* loops (one of them nested inside another) that can be parallelized. The first one is used only to initialize a matrix of Nx3. The second and third(nested) loops are where the entirety of the calculations happen, this implies that this block is where there are the most computations.

There a many ways to deploy for loop parallelization as the **Loop Schedueling** alters greatly the efficiency of the operation. The options made avaliable by the OpenMP are *static*, *dynamic*, *guided*, *runtime* and *auto*. Each type has its advantages and disadvantages which the group had to consider in order to best fit the use case.

Although the scheduling is important the most crutial part of this project is the prevention of data races. This error is caused by multiple threads accessing the same variable in an inordarly way compromising its values in the shared memory. Due to the thread not being correctly synchonized this may lead to unpredictable and erroneous behavior. To prevent this the group needs to verify one of two options:

- Wrapping Variables in a Private Scope: If the variable is only used inside the thread and its values are unimportant to the rest of the code, each thread can have its own registry of these values. This can be assured through the use of the *private* key word or the initialization of such values inside the thread its self.
- Supervision of the values: In case the variable is global or needed by multiple threads there is the option to signal the zone as critical, this means that there can only be one thread at a time executing the signaled code. This option can be done in OpenMP using the *atomic* or *critical* keyword. Although this practice corrects the data race it creates a bottle-neck as threads needs to wait to enter the critical zone.

  An alternative to this method is use the *reduce(func:var)* funtion that creates a private variable for each thread an as the threads the variables are “joined” using the function indicated.

3.3 | Our Approachs & Implementation

3.3.1 | First *for* Loop

Due to its low computational weight the group speculated that the use of threads on this block would be unproductive as the synchronisation overhead would surpass its gains.

3.3.1.1 | Scalability Analysis

Analysing the code, the group observed that in each iteration variables are independent and the workload uniform, this being said, implementing the first approach (parallelizing) would logically lead to a static schedule, which benefits uniform workloads. This approach is intrinsically associated to an execution overhead that scales as the units involved scale. After running some tests, the group observed that without parallelization the performance has a slight increase as proven by Figure 14.

3.3.1.2 | Implementation

In the first *for* loop, the group choose to keep the same sequencial code, given that based on our scalability analysis we concluded that the overhead created outweights the benefits of parallelization.

3.3.2 | Second & Third *for* Loop

Upon initial inspection, the nested loop exhibits imbalances, with each iteration successively reducing in size, consequently, later threads bear significantly lighter workloads

compared to their earlier counterparts. Static scheduling is therefore discouraged, prompting a recommendation for a thorough analysis of dynamic and guided scheduling methods.

The group also observed that all variables within the loop are susceptible to data races, with particular attention drawn to the "PE" and "a" variables, given their global scope.

Finally our group decide to parallelize the outter loop to reduce the synchronisation overhead that would happend at every iteration of the outter loop. This also increases the memory wall as parallelizing the outer loop allows for better cache utilization. Each thread can load a portion of the a and r arrays into cache and work on that portion for many iterations of the inner loop, reducing cache misses.

3.3.2.1 | Scalability Analysis

Dynamic and guided scheduling both involve dividing iterations into "chunks" assigned to threads upon completing previous chunks. The key difference is that guided scheduling reduces chunk size as execution progresses, promoting a more even load balance, especially in unbalanced loops. However, this reduction in chunk size increases the computational requirements for assigning threads. In our tests, dynamic scheduling showed a slight one-second improvement over the guided protocol, prompting our choice to be the implementation of dynamic scheduling.

3.3.2.2 | Implementation

Based on the Scalability Analysis, the group implemented the following techniques to address each problem. In the second *for* loop, dynamic scheduling was adopted along with the Reduction of incremental/decremental variables. In the third *for* loop, variables were declared within the private scope of each thread combined with a slight code restructure - to decrease cache reads and improve legibility.

3.4 | Results Analysis

In order to test the solution developed, the group tested executing with an increased number of cores in the cluster, where each core represents one thread.

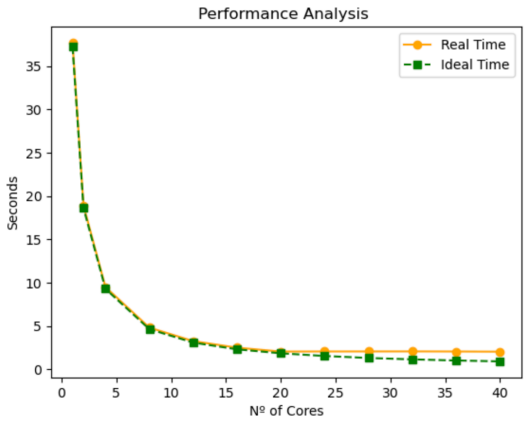

Figure 5: Plotting the Real vs Ideal Time

The Ideal Time is derived from dividing the sequential code execution time by the number of cores, serving as a theoretical benchmark. Meanwhile, the Real Time represents the results obtained from tests conducted by our team on the SeArch Cluster.

As depicted in the Figure 5, real-time performance aligns closely with ideal time, showing expected variance, until reaching a stage of uniform performance. This occurrence is elucidated by Amdahl's Law and the impact of parallelization overhead.

Amdahl's Law asserts that the speedup of a program is constrained by the portion of the code resistant to parallelization, which means the inherent sequential code imposes a performance ceiling. Additionally, the increase in core count can introduce overhead in terms of communication and coordination between threads. This overhead at some point counterbalances the gains achieved through parallelization.

3.5 | Conclusion Phase 2

On the second phase, the team's approach to parallelization stands as a robust method, effectively optimizing the performance of our sequential code, from 37 secs to 2.05 secs . The success of our optimization highlights the efficacy of OpenMP in enabling efficient parallel execution on the CPU. This part of the project not only served as a testament to the impactful implementation of targeted parallelization but also underscores the nuanced balance required between sequential and parallel execution for achieving optimal performance.

4 | Phase 3 - Implementing CUDA

In the third and final phase of this assignment, the group opted for the path of designing and implementing parallel code to be executed on accelerators. To achieve this, we chose to employ the CUDA API, a powerful framework tailored for programming on NVIDIA GPUs. In CUDA programming, parallelizable portions of code, often referred to as "kernels," are identified and offloaded to the GPU. These kernels are designed to perform computations concurrently on multiple data elements, taking advantage of the massively parallel architecture of GPUs.

To make this happen, our group used the final code from phase 1 as a starting point. So, the additional code we put in during phase 2, where we implemented OpenMP, doesn't play a role in what we're doing in this current phase. This way, we can focus on the specific goals of this phase without any interference from the previous changes.

4.1 | Profiling

The codebase at the beginning of this phase remains consistent with that of the start of phase 2, allowing the group to leverage the previously conducted code profiling. During the profiling, the function "joinedPotentialComputeAcc()" emerged as the discernible bottleneck in terms of execution time, as seen in Figure 12, pinpointing a key area for improvement and optimization.

4.2 | Approach

Code Analysis

Examining the *joinedPotentialComputeAcc()* function, and having worked with it in Phase 2, led the group to the unan-

imous conclusion that the entire code is amenable to parallelization.

Focusing on the code reveals a nested for loop structure, iterating through a matrix of size Nx3. Notably, within the inner loop (j), an interesting property is observed, as visualized in Figure 6. This visualization shows the amount of positions accessed by j in each iteration of i. Consequently, it becomes apparent that the final iterations are much lighter due to the amount of calculations done.

Other details worth meationing are :
- variable PE is a global var, set as zero in each cycle *i*.
- matrix a[N][3] is a global var used in every cycle of *j*.
- there's no need to initialize each a[N][3]

This implies that the results of the GPU computation need to be retrieved to the CPU at the end of the parallel code execution.

Designing the Solution

1st Solution

Initially, the group's instinctive solution involved assigning each thread to a specific row of the matrix and performing computations in parallel. Within each thread, the iteration over the values of j involved calculating the potential energy (P) and determining the corresponding forces to add/subtract at the array d_a[], representing particle accelerations.

Upon further consideration,as seen in Figure 6, as the iterations progress, the load carried by each thread decreases. This consequentially causes a big load imbalance in the initial threads compared to the later ones.

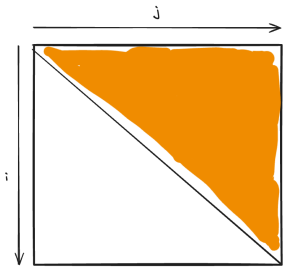

Figure 6: Threads Workload

2nd Solution

When designing an improved solution, the group aimed to mitigate the affore mentioned issue. Upon examination, the matrix can be divided in half, as shown in Figure 7, the second part of the workload can be seamlessly incorporated into the "empty" spaces of the threads in the first part, resulting in solution shown in Figure 8 - that ensures an even workload for each thread and half of total iterations.

In contrast to the previous solution, the updated approach introduces a few key differences. Notably, it involves incorporating the calculation of `inv_i`, that is used in the calculations as the index for the inverted section of the matrix, and reducing the value of `N` by half which also reduces the number of threads and blocks created. Additionally, a custom section is integrated within the inner loop. In this section, identical calculations are performed, but `rij[n]` now utilizes the computed value of `inv_i`.

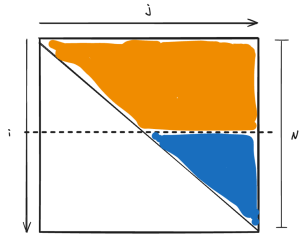

Figure 7: Spliting the matrix by half

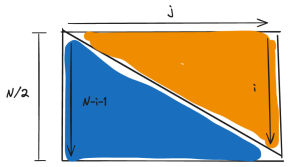

Figure 8: Balancing each thread workload

Implementation

In order to implement each solution designed, the group outsourced all the computation to the gpu, implementing the following CPU and GPU workflows:

CPU workflow - occurs inside the function *joinedPotentialComputeAcc()*
- allocate space in the GPU memory using *cudaMalloc()*
- if applied, copy data from CPU → GPU using *cudaMemcpy(dst,src,numByts,cudaMemcpyHostToDevice)*.
- if applied, set data allocated in GPU memory to zero(0) *using cudaMemset()*.
- launch the *kernel<<< grid_size, block_size>>>(args)*, function where
- copy data from GPU → CPU using *cudaMemcpy(dst,src,numByts,cudaMemcpyDevicetoHost)*.
- if applied, work on the data received.

GPU workflow - the kernel is defined on the function *rijCalc()*
- assign thread to each row in r[N*3] int i = threadIdx.x + blockIdx.x * blockDim.x
- Note: that the r matrix in the GPU is an array of size 3*N. Because of this the position calculation is diferent, the following shows the request of the same position of r:

$$r[i][k] == r[i * 3 + k] \tag{1}$$

- instantiate each var used in the local (thread) context.
- run the kernel
- sum all of the d_P values into potentials global variable
- Solution 1:
  - follows the flow of the sequential version
  - but has an atomic substraction for the a[j]
  - adds a[i] to local variables (a0, a1, a2) in order to reduce the amount of atomic operations
  - atomicly adds the local variables to d_a (return variable of a)
  - does the same two points for the Potential values as well
  - Potential is returned as an array where each value is a sum of the Potentials of the threads inside a block, this helps reduce the amount of conflicts but also doesn't postpone all of the aditions for the CPU

- Solution 2: This solutions follows exactly the same flow of the previous but instead of doing it only for the one iteration when j reaches the later values of the loop (j > inv_i) it does the solution 1 flow twice, using inv_i as the reference index.

Auxiliary Functions The group encountered challenges when using the default CUDA functions, *atomicAdd()* and *atomicSub()*, in the computations. These functions enable threads to increment or decrement the value of a shared variable atomically, preventing race conditions that may arise when multiple threads attempt simultaneous modifications to the same memory location.

In addressing these issues, the group undertook research and found a solution. This solution was implemented in the form of custom functions, *ourAtomicAdd()* and *ourAtomicSub()*, serving as alternatives to the default CUDA atomic operations for double values.

4.3 | Tests

As an additional requirement in this final stage, the group conducted a series of tests to thoroughly assess the performance of the developed implementation.

The chosen approach involved using diverse sets of inputs for each presented solution. This comprehensive testing aimed to determine the effectiveness of each solution in various scenarios and evaluate how well each scales. This also allowed the group to validate the best number of threads for the number of particles calculated.

Solution 1

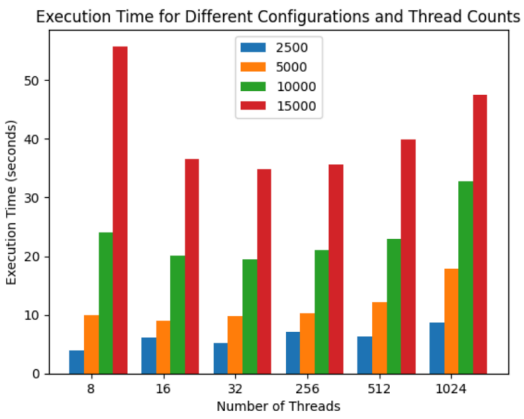

Figure 9: Bar Chart Solution 1

N	Sequential	CUDA(Solution 1)	SpeedUp(x)
2500	8.892	3.851	2.309
5000	35.557	8.968	3.965
10000	107.969	19.457	5.549
15000	320.155	34.842	9.189

Solution 2

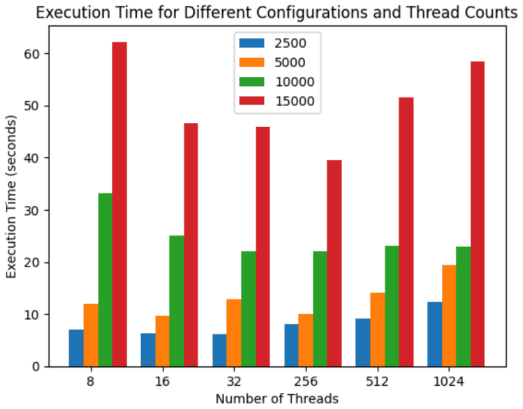

Figure 10: Bar Chart Solution 2

N	Sequential	CUDA(Solution 2)	SpeedUp(x)
2500	8.892	6.091	1.460
5000	35.557	9.689	3.670
10000	107.969	22.023	4.903
15000	320.155	39.555	8.095

4.4 | Results Analysis

Analyzing the test results, the optimal thread configuration in computational tasks is intricately linked to the value of N, the problem size. For low N, fewer threads suffice due to the lighter computational workload. Conversely, for high N, an intermediate thread count proves most efficient, balancing parallelism and computational overhead.

As N increases, the observed speedup escalates, underscoring the scalability of parallel processing for larger problem sizes. Notably, Solution 2 exhibits suboptimal performance, necessitating a reconsideration of the solution design approach - which made us concluded the following paragraph.

In GPU computing, load imbalance seems less critical when synchronization is not paramount, given the parallel architecture's adeptness at distributing tasks across numerous cores. Concurrently, the use of atomic functions, while ensuring data integrity, introduces a trade-off in increased wait times. Striking a balance between parallelism and minimizing synchronization overhead is pivotal for optimizing overall performance.

4.5 | Conclusion Phase 3

In conclusion, embracing the CUDA framework for the third phase presented both a challenge and a valuable learning opportunity. The technical insights gained throughout this journey served as a significant asset for future endeavours in prominent areas such as High-Performance Computing (HPC), Distributed Systems, and Artificial Intelligence (AI).

As a final note, the group considers the overall project structure proved to be challeging but adequate to the material taught in the practical classes, leading to a comprehensive learning experience that encompassed various tools and approaches essential for addressing various optimization problems.

5 | Attachments Phase 1 :

```
for (k=0; k<3; k++) {
    r2 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
}
```

Listing 1: Code from the *Potential()* before optimizing

```
r2 = (r[i][0]-r[j][0])*(r[i][0]-r[j][0])
+ (r[i][1]-r[j][1])*(r[i][1]-r[j][1])
+ (r[i][2]-r[j][2])*(r[i][2]-r[j][2]);
```

Listing 2: Code from the *Potential()* after optimizing

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 63.35     50.10    50.10      201   249.25   249.25  Potential()
 36.64     79.08    28.98      202   143.46   143.46  computeAccelerations()
  0.08     79.14     0.06      201     0.30   143.76  VelocityVerlet(double, int, _IO_FILE*)
  0.03     79.16     0.02      201     0.10     0.10  MeanSquaredVelocity()
  0.00     79.16     0.00     6480     0.00     0.00  gaussdist()
  0.00     79.16     0.00      201     0.00     0.00  Kinetic()
  0.00     79.16     0.00        1     0.00     0.00  initialize()
  0.00     79.16     0.00        1     0.00     0.00  initializeVelocities()
```

Figure 11: gprof profilling on the base code.

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
```

Listing 3: Code from *Potential()*

```
sigma6 = sigma*sigma*sigma*
         sigma*sigma*sigma
quot = sigma6 / (r2);
term2 = quot * quot * quot ;
term1 = term2 * term2;
```

Listing 4: Code simplified from *Potential()*

6 | Attachments Phase 2:

```
93.47%          8459  MDseq.exe  MDseq.exe         [.] joinedPotentialComputeAcc
 6.50%           581  MDseq.exe  [unknown]         [k] 0xffffffffa158cbc0
 0.00%             1  MDseq.exe  [unknown]         [k] 0xffffffffa1596f99
 0.00%             2  MDseq.exe  [unknown]         [k] 0xffffffffa0e63d10
```

Figure 12: Perf Profiling of the Sequential Code

```
37.239281036 seconds time elapsed

37.227118000 seconds user
 0.005000000 seconds sys
```

Figure 13: Execution time of the Sequential Code

```
 4.832925133 seconds time elapsed

38.603579000 seconds user
 0.003000000 seconds sys
```

Figure 14: Execution time with first *for* loop parallelized

```
void joinedPotentialComputeAcc(){
    for(i=0; i<N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    PE = 0.;

    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            rij[0] = r[i][0] - r[j][0];
            rij[1] = r[i][1] - r[j][1];
            rij[2] = r[i][2] - r[j][2];

          rSqd = sigma6 / (rij[0] * rij[0] + rij[1]
* rij[1] + rij[2] * rij[2]);

            term2 = rSqd * rSqd * rSqd ;
            term1 = term2 * term2;
            PE +=8*epsilon* (term1-term2);

            f = term2 * rSqd * (48 * term2 - 24);

            a[i][0] += rij[0] * f;
            a[j][0] -= rij[0] * f;
            a[i][1] += rij[1] * f;
            a[j][1] -= rij[1] * f;
            a[i][2] += rij[2] * f;
            a[j][2] -= rij[2] * f;
        }
    }
}
```

Listing 5: Sequential Code from *joinedPotentialComputeAcc()*

```
void joinedPotentialComputeAcc(){
    int i;
    double sigma6 = sigma*sigma*sigma
    *sigma*sigma*sigma;

    for(i=0; i<N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }

    #pragma omp parallel for schedule(dynamic, 40)
reduction(+:PE) reduction(+:a[:N][:3])
    for (i = 0; i < N-1; i++) {
        PE = 0.;
        for (int j = i+1; j < N; j++) {
           double f, f0, f1, f2, rSqd, term1, term2;
            double rij[3];
            rij[0] = r[i][0] - r[j][0];
            rij[1] = r[i][1] - r[j][1];
            rij[2] = r[i][2] - r[j][2];

           rSqd = sigma6 / (rij[0] * rij[0] + rij[1]
* rij[1] + rij[2] * rij[2]);
            term2 = rSqd * rSqd * rSqd ;
            term1 = term2 * term2;

            PE +=8*epsilon* (term1-term2);

            f = term2 * rSqd * (48 * term2 - 24);
            f0 = rij[0] * f;
            f1 = rij[1] * f;
            f2 = rij[2] * f;

            //  from F = ma, where m = 1 in natural
units!
            a[i][0] += f0;
            a[i][1] += f1;
            a[i][2] += f2;
            a[j][0] -= f0;
            a[j][1] -= f1;
            a[j][2] -= f2;
        }
    }
}
```

Listing 6: Parallel Code from *joinedPotentialComputeAcc()*